

Large-Scale Reasoning with OWL

Michael Ruster

University of Koblenz-Landau, Campus Koblenz

1 Introduction

In recent years, the Semantic Web has grown in size and importance. More and more knowledge is stored in machine-readable formats like RDFS or OWL. For many applications, knowledge extraction and reasoning is one of the core requirements. Through reasoning, knowledge can be logically derived that is not explicitly present in the data. Due to the complexity and amount of knowledge on the Semantic Web, this can easily become a difficult task. The bottlenecks are the time used for processing a query as well as the memory needed while reasoning.

This paper outlines common approaches for efficient reasoning on large-scale data. It therefore presents techniques implemented in reasoners, which are able to process billions (10^9) of triples. The paper focuses on OWL because it is widely used as a knowledge representation ontology language on the Semantic Web and because it is rich in features. First of all, a brief introduction to the Semantic Web is given in Section 2. For this, common properties of it are highlighted which may influence the choice of language selection for knowledge representation. Subsequently, Section 3 will give an overview of OWL and some of its sublanguages. The choice of sublanguages is shortly explained based on the requirements of modelling knowledge on the Semantic Web. Section 4 illustrates two main techniques for large-scale reasoning. Differences between the approaches are being highlighted. Furthermore, for both techniques, one example reasoner is presented together with some optimisation strategies they implement. Finally, Section 5 summarises this paper.

2 Semantic Web and Language Features

This section focuses on the Semantic Web and some of its most important properties. Thereby, it introduces the various important terms which are used throughout this paper. Furthermore, this section describes two different logics that can be of use for modelling Semantic Web data.

The Semantic Web's purpose is to extend the World Wide Web by encoding its information in machine-parseable ways [3]. As a result, it should be possible for machines to easily extract a Web page's "meaning" [31]. The Semantic Web is built using various technologies like *ontologies* to achieve this. Ontologies are conceptual models, encoding a set of terms and relationships between them [3]. This enables the organisation and exchange of information as well as reasoning

on it. Formal descriptions of ontologies are given through *ontology languages*. Ontology languages are using logics to express knowledge. OWL is built around description logics to which an overview is given in Section 4. Its statements can be expressed through a subject-predicate-object structure which are called *triples*. All statements that describe the taxonomy of the domain by expressing terminological knowledge are called the *TBox* of the ontology [2]. Likewise, all statements describing assertions about instances like their properties and relations to other instances are called the *ABox* [2].

Semantic Web data may originate from many diverse fields such as biology, medicine [31] or journalism like the New York Times¹. Hence, the knowledge to be represented is heterogeneous and may require different statement capabilities of its ontology language. A richer amount of features may also mean a higher complexity while reasoning [cf. 22]. Nevertheless, depending on the domain to model, more features might be needed. For example, generally, every statement is either true or false. But modelling uncertainty or imprecision is then not possible e.g. in classical description logic, a statement cannot express that a paper is “almost” finished. A solution for this is to introduce a finer range of truth values. This can be done e.g. with *fuzzy logic* by assigning such an additional truth value in the range of $[0, 1]$ to each triple [46].

Similarly, reasoning over incomplete and conflicting knowledge can be problematic for instance when trying to calculate a transitive closure over multiple triples. A common example expresses that birds can fly and that penguins are birds. But penguins form an exception regarding the capability of flight while *most* birds are actually able to fly. *Defeasible logic* [27] allows expressing sentences like “generally, birds can fly” while also modelling that penguins cannot although they are birds [cf. 24]. This logic introduces three main elements [27]:

Strict rules model knowledge that is true in all cases like “swallows are birds”.

Defeasible rules express that something “typically” holds like “birds can fly”.

They can be defeated by other defeasible or strict rules. Rules can be assigned priorities to determine which rule may defeat another one [27, 11].

Undercutting defeaters formulate possible exceptions to defeasible rules without them being expressive enough to allow any concrete inference. For example, “an injured bird *might* not be able to fly” would not allow an inference that all injured birds are unable to fly. Instead, it highlights that there might be exceptions which would make an inference impossible.

Fuzzy logic and defeasible logic may seem similar at first but are indeed different. As Covington [7] notes, fuzzy logic allows reasoning with a certain level of uncertainty or imprecision. Whereas defeasible logic ignores these degrees of truths and instead expresses that some rules may be overridden by others.

Both embody interesting concepts for the Semantic Web. Fuzzy logic can be used to express various level of trust in certain sources. For example, an article by a renowned newspaper will often be more trustworthy than one by a rather unknown personal blogger. Defeasible logic helps modelling conflicting knowledge

¹ <http://data.nytimes.com/> — last accessed 25 March 2015, 11:00

in a different way. In the example before, the information by the blogger can be defeated by the one published by the newspaper. It remains unclear how much more credible the newspaper is compared to the blogger. Yet, the information from the newspaper article can be seen as truth instead of only information closer to the actual truth. This allows modelling of incomplete knowledge with knowledge capturing typical states and being defeated by more specific rules. Hence, when knowledge changes would appear that are expressed through defeasible rules, these defeasible rules could immediately be added without entailing further processing [11].

It depends on the ontology developers whether they see a need for these logics to describe their data. In some contexts, it can be useful. However, it always adds complexity to the reasoning process. In the next section, sublanguages of OWL are discussed to illustrate possible choices in expressiveness that are already provided by OWL without any further extensions.

3 OWL and its Sublanguages

This section gives a quick introduction to OWL and its sublanguages in terms of their features. The purpose of this section is to illustrate that the choice of an ontology language is an important decision for reasoning on big data. For this purpose, reasoning complexities of most languages are given. Their most important properties are presented and thus it should be understood, why some languages are commonly used for this type of reasoning.

The *Web Ontology Language* (short: *OWL*) is a specification by the W3C with the purpose of representing knowledge in machine-parsable ways. In the OWL 1 Web Ontology Language Guide [33], the W3C explains that OWL 1 can be divided into its three sublanguages *OWL 1 Full*, *OWL 1 Lite* and *OWL 1 DL*. These sublanguages themselves can be further divided and differ in their expressiveness with OWL 1 Full being the feature-richest. It can be seen as an extension of RDF, which includes meta-modelling capabilities of RDF Schema (short: *RDFS*) [25]. OWL 1 Lite and OWL 1 DL on the other hand can be understood as extensions of a subset of RDF [25]. Ontologies created using OWL 1 Lite are subsets of those built with OWL 1 DL which themselves are subsets of OWL 1 Full ontologies. Detailed differences can be taken from the OWL 1 semantics specification [29].

With rising amount of features, the complexity of reasoning increases as well: Reasoning in OWL 1 Lite is complete for EXPTIME, in OWL 1 DL for NEXPTIME and OWL 1 Full is even undecidable [17]. The revision of OWL 1, called OWL 2, even extends the feature set of the sublanguages, resulting e.g. in a reasoning complexity of 2NEXPTIME for OWL 2 DL [13]. Thus, especially when reasoning on billions of triples, it is important to decide on using the least complex sublanguage that offers all needed features for the given modelling purpose.

Commonly used for modelling knowledge while keeping reasoning feasible are sublanguages of OWL 2 DL, also referred to as *profiles*. These profiles are characterised by further restricting the feature set and hence making reasoning

more feasible [18, 22]. Hereby, these ontology languages also become simpler to implement, extend and easier to understand [22]. The W3C distinguishes three OWL 2 profiles, namely *OWL 2 QL*, *OWL 2 RL* and *OWL 2 EL* [26]. These profiles have a reduced reasoning complexity of as low as PTIME for OWL 2 RL [cf. 26, 6]. One of the restrictions that all the profiles share is that for axioms which define subclass inclusion, they disallow unions of classes. Otherwise, reasoning on these ontology languages would become NP-hard already. Similarly, all profiles forbid the use of negations and universal quantifiers on the left-hand side. Their concrete restrictions are explained in the OWL 2 profile specification [26] and are outside of this paper’s scope. However, Krötzsch [22] summarises their most important characteristics as follows:

OWL 2 QL was designed as a query language to ease information retrieval from ontologies as a database. Hence, queries may extract matching data together with facts inferred from it in LSPACE [26]. This sublanguage’s design enables *query rewriting*, where the query is rewritten e.g. into an SQL query that can be directly executed on an SQL database [4]. The technique of query rewriting is discussed in more detail in Section 4.2 with an approach being presented where the query is split up into related rules that eventually may return instances.

OWL 2 RL is often applied to contexts where the TBox is notably smaller than the ABox, which is common for the Semantic Web. It was developed as a scalable solution and allows querying on big datasets while retaining most of OWL 2 DL’s expressiveness [26].

OWL 2 EL is likewise aimed at scenarios with big ABoxes [26]. Thus, it is used e.g. for biomedical ontologies like Systematized Nomenclature of Medicine Clinical Terms² (short: *SNOMED CT*), which gathers information about human diseases. It is the most restricted profile and therefore the least expressive.

Besides the OWL 2 profiles, *OWL pD** (also known as *OWL Horst*) is also often used for reasoning on huge amounts of data. It is an extension of RDFS combined with a subset of OWL 1 [18]. As a result, it becomes more expressive than RDFS while remaining less computationally complex than OWL 1 Full [40]. Reasoning in OWL pD* has a complexity of NP and P for some special cases [18]. Urbani et al. [40] describe it as “a de facto standard for scalable OWL reasoning” [40, p.216]. The development of OWL 2 RL was partly influenced by OWL pD* [26]. Thus, both are frequently used for large-scale reasoning [e.g. 4, 15].

Regardless of the revision of OWL, it can be said that OWL Lite is not expressive enough for modelling most knowledge on the Semantic Web. OWL DL and OWL Full however are not tractable for reasoning on huge datasets. Therefore, restricted sublanguages are selected for knowledge modelling and reasoning on the Semantic Web. OWL pD* and OWL 2 RL are the languages most commonly applied to this task, often using a materialisation strategy presented later in Section 4.1. Nevertheless, other profiles such as OWL 2 QL are also used in some reasoners following a query rewriting strategy as further explained in Section 4.2.

² <http://www.ihtsdo.org/> — last accessed 06 March 2015, 16:30

4 Approaches for Large-Scale OWL Reasoning

First of all, this section gives an overview of description logics, focussing on one concrete language as an example. The description logics syntax and semantic will be used when subsequently presenting the two inference methods *forward chaining* and *backward chaining*. The later sections go into detail about implementations of these methods. Section 4.3 will furthermore give an overview of a selection of OWL and/or Semantic Web reasoners.

Description logics are a first-order logic subset and allow the modelling of knowledge [2]. They are the foundation of OWL and other ontologies. A common example of description logics is the *Attributive Language with Complements* (short: *ALC*) [30]. It consists of *concepts*, *individuals* and *roles*. A concept can e.g. be a *Human* or *Building*. Individuals can be seen as elements of concepts e.g. *Peter* or *EiffelTower*. A function—also called an *interpretation*—associates the individuals with concepts, so that for example *Peter* : *Human* expresses that the individual *Peter* satisfies the *Human* concept. Examples for roles are *hasChild* or *builtBy*.

To link these elements with each other, *ALC* introduces *connectives*. One such connective is the concept inclusion “ \sqsubseteq ”, which allows expressing that one concept is more general than the other. For example *Woman* \sqsubseteq *Human* may model that the gender neutral concept of a human includes women. Like in set theory, concepts can also be unified so that e.g. *Woman* \sqcup *Man* \sqsubseteq *Human*. The intersection of concepts indicated by “ \sqcap ” is included analogously. There is also a top concept “ \top ” which is the most universal concept and thus subsumes all others. Likewise, the bottom concept “ \perp ” models the concept of nothingness. An example, using the negation connective “ \neg ”, would be *Woman* \sqcap \neg *Woman* \sqsubseteq \perp . Existential and universal quantifiers—“ \exists ” and “ \forall ” respectively—allow expressing knowledge including roles. For example, “a building is built only by humans or no one” can be modelled as *Building* \sqsubseteq \forall *builtBy.Human*. Similarly, modelling a parent as someone who has had a child can be expressed with *Parent* \sqsubseteq \exists *hasChild.T*. The given introduction to *ALC* and description logics in general is incomplete and the interested reader may consult Schmidt-Schauß and Smolka [30] and Baader [2] respectively.

Forward and backward chaining differ in the “direction” of reasoning. Reasoning is used to derive implicit knowledge from ontologies by applying terminological knowledge to the explicitly modelled data [2]. Forward chaining is *data-driven* meaning that reasoning will start from existing data and infer new knowledge as long as it is possible [32]. Given are two example class subsumption rules in Equation 1 and 2.

$$X \sqsubseteq P \tag{1}$$

$$Z \sqcup P \sqsubseteq Y \tag{2}$$

A forward chaining approach searches for rules with matching antecedents and then assumes true consequents. Equation 3 shows how example data may express

a being of class Z and X . Next, it can be matched against the antecedents of the example rules and hence reasoned to being of class Y .

$$a : Z \sqcup a : X \xRightarrow{(1)} a : Z \sqcup a : P \xRightarrow{(2)} a : Y \quad (3)$$

Backward chaining on the other hand is *goal-driven* [32]. It divides a goal into smaller subgoals and tries to resolve those. By matching rules for true consequents and then assuming true antecedents, backward chaining reasons for data matching the initial goal. Likewise, Equation 4 illustrates a backward chaining application assuming a being of class Y .

$$a : Y \xRightarrow{(2)} a : Z \sqcup a : P \xRightarrow{(1)} a : Z \sqcup a : X \quad (4)$$

The following sections explain how the approaches can be used for Semantic Web reasoning and what advantages they have over the other. Section 4.1 presents a common way to use forward chaining for this task by also presenting a typical programming model and an overview of an implemented reasoner. Subsequently, Section 4.2 discusses the application of backward chaining to reasoning on billions of triples. Analogously, the main properties and approaches of an implementation are shown.

4.1 Materialisation

Materialisation is a forward chaining approach [21]. The idea behind is to compute all inferences prior to reasoning and storing them for later querying [32]. In this section, after discussing the most important advantages and disadvantages to this technique, a common programming model called *MapReduce* is explained. Furthermore, a reasoner using MapReduce is presented.

Computing all inferences first once, allows fast query answering as it is comparable to lookups in a database. On the other hand, the initial materialisation process is time and memory consuming. As an example, `owl:sameAs` is one of the most commonly found axioms according to Hogan et al. [14]. It is used to express equivalent individuals. If a reasoner would be naïvely implemented, a full closure would be in $\mathcal{O}(n^2)$. On a corpus containing 33,052 equivalent individuals, Hogan et al. [16] prospected 1,092,434,704 triples and an additional two billion for those individuals being included in other statements. Another downside is that materialisation must be done anew every time the data is updated. Due to the nature of the Semantic Web, data may change frequently and hence regular updates are necessary to ensure recent results.

MapReduce MapReduce is a programming model developed by Dean and Ghemawat [8] for Google Inc. with the goal to process big amounts of data efficiently. It tries to achieve this by allowing distributed and parallel data processing and thus reducing the load on single machines/cores. The most frequently used implementation is Apache Hadoop³. MapReduce can also be

³ <https://hadoop.apache.org/> — last accessed March 08 2015, 17:30

used for reasoning on Semantic Web data [cf. 36, 42] for which it then becomes a forward chaining approach. The process is split into three consecutive steps as described by Dean and Ghemawat [8]:

1. **Map** is the first phase and describes the act of pre-processing input data as a list of key-value pairs. The result of every processed key-value pair is immediately emitted. Thus, this process may be parallelised and can hence speed up data processing. Its method signature can be given as:

$$\text{Map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$$

The different indices express potentially different datatypes.

2. **Shuffle** collects all results of the first phase and groups them by their key. Hence, the datatypes remain the same and a method signature can be given as:

$$\text{Shuffle}(\text{list}(k_2, v_2)) \rightarrow \text{list}(k_2, \text{list}(v_2))$$

3. **Reduce** is the phase in which the values with the same key are being processed. Values are being merged together and returned as a smaller result set. To avoid having to load the complete data into memory, the reduce phase is only given an iterator over the values. The list of returned values is of the same datatype as the intermediate values returned by the mapping phase. Its signature can be given as:

$$\text{Reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)$$

However, the commonly used implementation Apache Hadoop is more liberal concerning the domain of return values, by having the following method signature [1]:

$$\text{Reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(k_3, v_3)$$

Similar to the map phase, the reduce phase is generally executable in parallel.

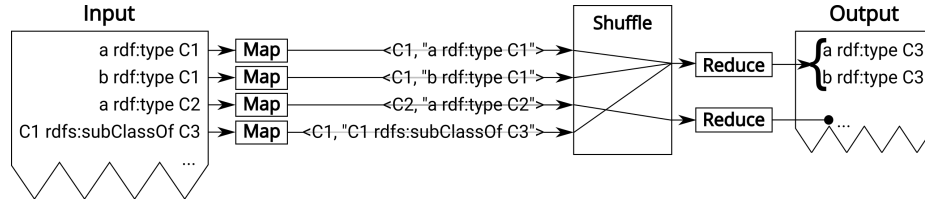


Fig. 1. An example MapReduce process for the rule $s \text{ rdf:type } x, x \text{ rdfs:subClassOf } y \Rightarrow s \text{ rdf:type } y$. The figure was taken from [42] and modified to explicitly include the shuffle phase.

Figure 1 illustrates a MapReduce application on an ontological rule. In this example, the two triples of the rule's antecedent share the common variable x . Hence, it is selected as key for the map phase. After this pre-processing, the shuffling groups all results by key. The reduce phase then produces the output drawn from the intermediate results matched against the consequent. At this

point, the full reasoning process may not already be finished. The reason for this is that the inferred knowledge matches part of the rule’s antecedent anew.

MapReduce applied to reasoning is not a trivial task which needs only one run as illustrated by the rule in Figure 1. Instead, there are multiple difficulties when trying to implement this approach in a way that is relatively light on space and time consumption. A few of these are presented in the next section together with an example MapReduce algorithm.

WebPIE as an Example Implementation of MapReduce Urbani et al. [41] employ MapReduce in their reasoner *Web-scale Inference Engine* (short: *WebPIE*) for reasoning on OWL pD* rules. The authors discuss the shown problem as a *fixed-point iteration*, expressing that the MapReduce process must be repeated until no more new triples are returned. This results in an additional problem as duplicates may be generated by having to apply some rules multiple times. Besides trying to generate as few duplicates as possible throughout the reduce phase, an additional MapReduce step is added that solely matches duplicates and removes them [39]. In the map phase, the step iterates over all triples and returns these triples as key, which removes duplicates in the succeeding shuffle phase. Then, in the reduce phase, triples are only emitted if they were inferred to distinguish them from the original input triples.

In general, Urbani et al. [42] try to execute as few MapReduce steps as possible. Yet, the processing of some rules together with others is either not possible or would have a heavy impact on performance. Thus, the authors implement various MapReduce steps which focus on certain rules.

One, which will be discussed as an example, is the algorithm shown in Listing 1.1. It calculates subclass relations and is separated into a map and a reduce method as shown in Line 1 and Line 10 respectively. The algorithm operates on triples having either `rdf:type` or `rdfs:subClassOf` as predicate. Depending on their predicate, the map phase uses different flags as keys—in this case either 0 or 1—together with the triple’s subject. The value associated with the key is the input triple’s object. Considering an example input triple `_x rdfs:subClassOf _y`, then the output’s key flag will be 1, its subject `_x` and the value `_y`.

The reduce method first removes all duplicates within the values. For the sake of simplicity, these values will be referred to as classes. In Lines 14 and 15 all TBox triples matching the classes as a subject are loaded into memory. These correspond to all the classes’ superclasses. Continuing the earlier example, having a triple `_y rdfs:subClassOf _z` in the TBox results in `_z` being added to the list of superclasses.

The reduce method distinguishes the input values based on their formerly associated predicate in the original triple as indicated by the key’s flag. In both cases, the method iterates over all superclasses. It checks that the currently processed superclass is not already contained in the set of classes to prevent duplicates. Whenever this is ensured, the method returns a new triple associated with a key. As the key is irrelevant in this scenario, the pseudocode reduce method

returns `null`. The triple models that the original input's subject is either of the same type as a matching superclass or also a subclass of that superclass. Thus, inferred triples will be returned that respect the class hierarchy. In the formerly given example, this would be `_x rdfs:subClassOf _z`.

```

1  map(key, value):
2      // key: source of the triple (irrelevant)
3      // value: triple
4      if (value.predicate == "rdf:type"):
5          key.flag = 0
6      if (value.predicate == "rdfs:subClassOf"):
7          key.flag = 1
8      key.subject = value.subject
9      emit(key, value.object)

10 reduce(key, iterator values):
11     // key: flag + triple.subject
12     // iterator: list of classes
13     values = values.unique // filter duplicate values

14     for (class in values):
15         superclasses.add(subclass_schema.getSuperclasses(class))

16     switch (key.flag):
17         case 0: // we're doing rdf:type
18             for (superclass in superclasses):
19                 if !values.contains(superclass):
20                     emit(null, new Triple(key.subject, "rdf:type", ←
21                                     ↪ superclass))
22         case 1: // we're doing rdfs:subClassOf
23             for (superclass in superclasses):
24                 if !values.contains(superclass):
25                     emit(null, new Triple(key.subject, ←
26                                     ↪ "rdfs:subClassOf", superclass))

```

Listing 1.1. The algorithm to do RDFS subclass reasoning initially presented by Urbani et al. [42]. This listing features a corrected and simplified version including changes from Urbani [39].

Urbani et al. [41] present approaches for tackling the difficulties they faced throughout building WebPIE. However, their solutions are mostly tightly coupled with the ruleset of OWL pD*. Nevertheless, they also propose two more generic optimisation strategies.

As the TBoxes are commonly a lot smaller than the ABoxes when reasoning with data from the Semantic Web, the TBoxes can often times be fully loaded into memory. This was also done in the algorithm shown in Listing 1.1. The resulting advantage is that then the triples of instances can be streamed and thus directly processed. But, Urbani et al. [41] note that this is not possible for

rules which require joins between multiple instance triples in their antecedent. Equation 5 shows such a rule from the OWL pD* ruleset.

$$v \text{ owl:sameAs } w, w \text{ owl:sameAs } u \Rightarrow v \text{ owl:sameAs } u \quad (5)$$

There, for both triples in the antecedent, matching instances in the ABox must be looked up each. Again, the authors propose a solution concretely fit to the affected OWL pD* rules.

However, they also describe a common technique for reducing the overhead created by rules considering the `owl:sameAs` axiom as in Equation 5. The input is modified so that synonymous instances are replaced by a unique identifier representing an entire equivalence class each [e.g. 40, 5]. As a result, the required space and computation time are both reduced drastically [23].

Although having only highlighted a few bottlenecks, it should have become clear that applying MapReduce to reasoning on billions of triples is a complex task. Albeit there being some reoccurring problems shared by most OWL sublanguages, many optimisation approaches are tailored to concrete rules. Therefore, there is a lack of efficient universal solutions.

4.2 Backward Chaining

With backward chaining, reasoning is done at runtime, once the query is posed. Hence, no prior computation is needed.

Backward chaining has two advantages over materialisation as described by Urbani et al. [44]. First, there is no need for precomputation due to the runtime reasoning. Neither is there generally a need for computing a full closure because any reasoning solely needs to be done as far as it is required to answer the query. As one result, an application exclusively using backward chaining may instantly be usable without prior time- and space-consuming computation.

Second, the results to a query consider recent data modifications. That is, when changes happen to the data such as deletions or additions, they are instantly retrievable through the reasoning done by query rewriting. Furthermore, this means that after changes have been applied to the data, no computationally and memory-intensive recomputation has to be done.

However, backward chaining also faces a great disadvantage compared with materialisation. As reasoning must be done with every new query, answering these is often time-consuming. Whereas for materialisation query matching instances may directly be returned from the fully reasoned knowledge base.

Query Rewriting When using backward chaining for reasoning on ontologies, some form of query rewriting is used. The concept of query rewriting is to reformulate the query into a query that respects the ontology's terminological axioms and retrieves the matching instances [19]. There are various options concerning how a query can be transformed such as logical rewriting [12] or even concrete SQL query rewriting [4]. Both are actively used in the context of *ontology-based data access* (short: *OBDA*).

OBDA describes the idea of storing ABoxes in traditional databases like relational database management systems while allowing the use of ontologic constraints [12]. Here, OWL 2 QL is often the preferred language as it was designed implicitly with OBDA as intended purpose [26, 20]. With OBDA, data access is offered through the ontology as an intermediate layer so that the queries are independent from the actual data storage [45]. This allows a unified semantical access to different data sources. Therefore, scalable, OBDA-enabled reasoning may become of special interest for Semantic Web reasoning due to its data being highly diverse.

Despite the existing potential, there are yet only few reasoners using query rewriting while scaling up to billions of triples. There is especially a lack of reasoners that focus on implementing the concept of OBDA and pertain this scalability. Thus, the subsequent section presents a reasoner that is not build around OBDA but instead uses an illustrative query rewriting approach while supporting reasoning on billions of triples.

QueryPIE There is no prevalently used programming model for query rewriting like there is MapReduce for materialisation. Urbani et al. [44] do query rewriting for OWL pD* by building a reasoning tree with the query being its root and the matching data its leaves. In backward chaining fashion, the query—an input triple pattern—is matched against rule consequents and their antecedents will then be used as new query triples. Figure 2 depicts a reasoning tree built from an example query `?S rdf:type Person`. The branches containing rules are connected by

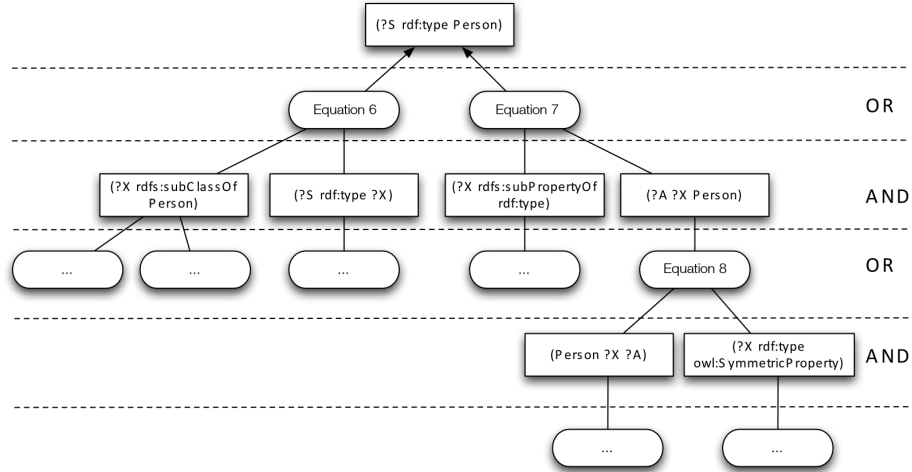


Fig. 2. An example of a reasoning tree [44] with the rule names being replaced to match this paper's equation numbers.

logical ORs whereas the triples are connected with ANDs. This way, all instances from leaves may be returned that match the rule of their parent node. However, it is not necessary for instance triples to match the rules of other branch nodes. Thus all matching triples may be returned, disregarding through which rules they can be inferred. In the given example, the consequents of Equations 6 and 7 match the input query.

$$s \text{ rdf : type } x, x \text{ rdfs : subclassOf } y \Rightarrow s \text{ rdf : type } y \quad (6)$$

$$s \text{ p } o, p \text{ rdfs : subPropertyOf } q \Rightarrow s \text{ q } o \quad (7)$$

Their antecedents are bound accordingly, substituting all variables where possible. In the antecedent $s \text{ p } o$ of Equation 7 for example, o is bound to **Person**. This allows to apply Equation 8 so that all variables from its antecedents are bound and reasoning can be continued on these.

$$p \text{ rdf : type owl : SymmetricProperty}, v \text{ p } u \Rightarrow u \text{ p } v \quad (8)$$

The tree building finishes when each branch has been built. A branch is built as deep as there are yet rules applicable or matching instances can be retrieved from the knowledge base. This process may consume a lot of time and also space. Thus, Urbani et al. [44] propose two optimisation strategies:

Precomputation of frequently appearing branches.

Tree-pruning describes the early discovery of branches which will not return any results and hence stopping to further follow them.

When implementing precomputation, its effectiveness is defined by how regular the selected branches appear. Urbani et al. [44] focus their optimisation on what they call *terminological triple patterns*. They define them as triple patterns whose object or predicate is a term from the RDFS or OWL vocabulary that they operate on. An example is given in Equation 9.

$$?X \text{ rdfs : subPropertyOf rdf : type} \quad (9)$$

Due to the commonly smaller TBox in Semantic Web data, there are few such terminological triple patterns. Yet, they affect many queries. As using forward chaining for precomputing would require the calculation of a full closure, backward chaining is applied on the selected terminological triple patterns. For this, a tree is built bottom up by iterative backward chaining as illustrated in more detail in [44].

The advantage of exclusively reasoning at runtime, is lost when using precomputation. However, it is still a much faster process than full materialisation. It especially remains fast enough to allow for frequent updates as typically found in Semantic Web data. However, this approach can hence no longer be classified as pure query rewriting but is actually a hybrid approach [43].

```

1 | reason(pattern):
2 |   // Get rules where the query pattern is more specific  $\leftarrow$ 
   //  $\hookrightarrow$  than the rule's consequent:
3 |   Rules applicableRules = ruleSet.applicable(pattern)

4 |   Results results = {}
5 |   for (rule in applicableRules):
6 |     antecedents = rule.instantiateAntecedents(pattern)
7 |     // Perform reasoning to fetch all antecedents:
8 |     for (antecedent in antecedents):
9 |       if (antecedent != terminological)
10 |        // Recursive call to the reasoner:
11 |        antecedents.add(reason(antecedent))
12 |        antecedents.add(KnowledgeBase.read(antecedent))
13 |        // Apply the rule using the antecedents triples:
14 |        results += rule.applyRule(antecedents)

15 |   return results

```

Listing 1.2. The reasoner which operates on the precomputed terminological triple patterns as given by Urbani et al. [44] with small modifications to match this paper's pseudocode style.

Listing 1.2 shows the algorithm in pseudocode used by Urbani et al. [44] to reason after precomputing has been completed. Hence, they call it the *terminological independent reasoner*. In the first step, all rules are retrieved whose consequent is more general than the query pattern. This corresponds to the OR-levels of the reasoning tree. While iterating over all rules, each rule's variables are subsequently bound to any matches from the query pattern as shown in Line 6. In the second loop all antecedents of a rule are processed which corresponds to the AND-levels of the reasoning tree. For this, the terminological independent reasoner is recursively called to further construct the branch. Due to precomputing, this step is only needed if the currently processed antecedent is not terminological as checked in Line 9. Next, a lookup in the knowledge base is done which also includes the already computed terminological triple patterns. After every branch of a rule have been computed, the rule is applied with all the previously computed antecedents. In the end, all applied rules will be returned, meaning that all of their variables are bound to match the query pattern.

As a second optimisation strategy, tree-pruning is used. It is built upon the formerly precomputed terminological triple patterns. Whilst backward chaining, rules, whose antecedent matches such terminological triple patterns, are prioritised. If these patterns do not return any instances, the rule will not be applied as they are more specific than the patterns. The processing of branches that are known not to return any results, will therefore be stopped early. Thus, a lot of unnecessary reasoning can be prevented.

In the next section, a selection of reasoners using forward and/or backward chaining is presented. However, all of the reasoners using backward chaining approaches either do not scale well or only partially support an OWL sublanguage.

QueryPIE is likely to be the first reasoner scaling to billions of triples while reasoning at runtime [43].

4.3 Other Reasoners

Besides WebPIE and QueryPIE, there are several other reasoners using forward and/or backward chaining. This section gives an overview to a selection of available reasoners.

OWLIM [21] is a semantic repository allowing the storage of and reasoning on knowledge bases by employing full materialisation. Bishop and Bojanov [4] state that OWLIM is capable to work on various rulesets including OWL pD* and OWL 2 QL as well as OWL 2 RL. OWLIM is divided into the free-for-use SwiftOWLIM and the commercial BigOWLIM [5]. Bishop et al. [5] state that SwiftOWLIM was developed for in-memory reasoning with smaller datasets. According to the authors, BigOWLIM on the other hand was developed for reasoning on billions of triples. For this, various optimisations were needed of which one is a special treatment of the `owl:sameAs` axiom. BigOWLIM uses a canonical representation for each equivalence class [5] similar to WebPIE's approach. Furthermore, it employs a backward chaining approach on data deletion to prevent a new full materialisation [5]. According to Bishop et al. [5], its use allows BigOWLIM to be applicable to frequently updated data as is typical for data from the Semantic Web.

Another, yet non-commercial, reasoner supporting multiple profiles is TrOWL [38]. It is an interface for multiple reasoners such as Quill or Pellet. Besides the support of OWL 2 QL and OWL 2 EL there is also partial support for tractable reasoning with OWL 2 DL [28]. Using reasoners like Pellet allows full OWL 2 DL reasoning support but reasoning is then no longer in polynomial time [9]. Reasoning of OWL 2 QL is done by using backward chaining, namely query rewriting [38]. OWL 2 EL on the other hand is reasoned in a forward chaining manner [28].

Tachmazidis et al. [36] presented a reasoner which employs Apache Hadoop for materialisation of knowledge that uses rulesets implementing defeasible logic. Their motivation for developing a reasoner on defeasible logic was to create inconsistency-tolerant reasoning that was able to deal with data of poor quality. Although the reasoner does not operate on OWL but on RDF data, this concept may have a high relevance for the Semantic Web as data from different sources are likely to be of different quality. Making use of parallelisation through MapReduce, Tachmazidis et al. [35] were able to build a reasoner scaling to billions of triples.

Stoilos et al. [34] developed a reasoner for an extension of OWL pD* that supports fuzzy logic. This allows expressing vagueness as already described in Section 2. Their work also uses the MapReduce implementation of Apache Hadoop. The authors followed the optimisation strategies of WebPIE and adapted them whenever necessary to support fuzzy logic. As a result, Stoilos et al. [34] claim to have created a fuzzy reasoner with a performance comparable to WebPIE.

Virtuoso Universal Server⁴ is a Web server and triple store. It is furthermore an OWL reasoner which allows backward as well as forward chaining [32, 37]. But, its support is restricted to a subset of OWL 2 RL [43]. Moreover, except for an RDF reasoner implementation [10], there do not seem to be any implementations targeting Web-scale triples.

Apache Jena⁵ is a framework written in Java to support building Semantic Web applications. Its included OWL reasoner supports forward and backward chaining as well as a hybrid between the two [32]. However, throughout the research for this paper, no implementation using the Jena reasoner could be found which would scale to allow reasoning on billions of triples.

Another reasoner is F-OWL [47] which primarily uses backward chaining. The authors call their speed-up strategy *tabling*. Their approach is to store the results of already reasoned triples and look them up whenever possible. As a result, the first queries are processed slowly but the system becomes increasingly faster on average for subsequent queries. However, it only supports OWL Full while neither being complete nor decidable. According to its website⁶, F-OWL has not been updated since 2003. Additionally, according to its authors [47] it does not scale and is thus unsuitable for reasoning on billions of triples.

5 Conclusion

The grand challenge of large-scale reasoning is to effectively use and reduce the time and space consumption. Essentially, the approaches do not differ from regular reasoning methods. Yet, the performance optimisation is crucial.

Additionally, the choice of ontology language may have heavy impact on the complexity of reasoning. OWL pD* and OWL 2 RL are commonly used for forward chaining approaches whereas OWL 2 QL is mainly used in backward chaining reasoners. Furthermore, when reasoning on the Semantic Web, extensions of the languages can be of interest. For example, using defeasible logic may allow reasoning with data of poor quality from different sources. Likewise, OBDA may be a suitable technology for working and reasoning on Semantic Web data.

Reasoning on big knowledge bases modelled in OWL is still rarely done using backward chaining. Instead, materialisation is the most common approach. However, materialisation alone cannot fit the Semantic Web's quickly changing nature. Thus, future research should consider large-scale reasoning using a hybrid approach of backward and forward chaining. Moreover, a requirement analysis of features needed for properly modelling and reasoning on Semantic Web data could be helpful. Due to the diversity of data, a prior separation by topics would probably be beneficial.

⁴ <http://virtuoso.openlinksw.com/> — last accessed 18 March, 22:00

⁵ <https://jena.apache.org/index.html> — last accessed 18 March, 08:30

⁶ <http://fowl.sourceforge.net/> — last accessed 17 March 2015, 22:00

Bibliography

- [1] Apache Software Foundation: Apache Hadoop 2.6.0 - MapReduce tutorial. <https://hadoop.apache.org/docs/r2.6.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html> (Nov 2014), accessed: Jan 28 2015, 20:30
- [2] Baader, F.: The description logic handbook: theory, implementation, and applications. Cambridge university press (2003), <http://books.google.com/books?id=riSeOKw5I6sC>
- [3] Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific american 284(5), 28–37 (2001), http://iscl2918929391.googlecode.com/svn-history/r347/trunk/RPC/Slides/p01_theSemanticWeb.pdf
- [4] Bishop, B., Bojanov, S.: Implementing OWL 2 RL and OWL 2 QL Rule-Sets for OWLIM. In: OWLED. vol. 796 (2011), http://webont.com/owled/2011/papers/owled2011_submission_3.pdf
- [5] Bishop, B., Kiryakov, A., Ognianoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: A family of scalable semantic repositories. Semantic Web 2(1), 33–42 (2011), <http://iospress.metapress.com/index/443N1L4245205818.pdf>
- [6] Cao, S.T., Nguyen, L.A., Szalas, A.: On the Web ontology rule language OWL 2 RL. In: Computational Collective Intelligence. Technologies and Applications, pp. 254–264. Springer (2011), http://link.springer.com/chapter/10.1007/978-3-642-23935-9_25
- [7] Covington, M.A.: Defeasible logic on an embedded microcontroller. Applied Intelligence 13(3), 259–264 (2000), <http://link.springer.com/article/10.1023/A:1026520227851>
- [8] Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6. OSDI'04, vol. 51. USENIX Association, San Francisco, CA (2004), <https://dl.acm.org/citation.cfm?id=1251264>
- [9] Dentler, K., Cornet, R., Ten Teije, A., De Keizer, N.: Comparison of reasoners for large ontologies in the OWL 2 EL profile. Semantic Web 2(2), 71–87 (2011), <http://iospress.metapress.com/index/N434780643M2N3U6.pdf>
- [10] Erling, O., Mikhailov, I.: Towards web scale RDF. Proc. SSWS (2008), <http://www.csee.umbc.edu/courses/graduate/691/spring12/03/papers/V0SArticleWebScaleRDF.pdf>
- [11] García, A.J., Simari, G.R.: Defeasible logic programming: An argumentative approach. Theory and practice of logic programming 4(1+ 2), 95–138 (2004), http://journals.cambridge.org/abstract_S1471068403001674
- [12] Gottlob, G., Orsi, G., Pieris, A.: Ontological Queries: Rewriting and Optimization (Extended Version). arXiv:1112.0343 [cs] (Dec 2011), <http://arxiv.org/abs/1112.0343>, arXiv: 1112.0343

- [13] Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P., Sattler, U.: OWL 2: The next step for OWL. *Web Semantics: Science, Services and Agents on the World Wide Web* 6(4), 309–322 (2008), <http://www.sciencedirect.com/science/article/pii/S1570826808000413>
- [14] Hogan, A., Delbru, R., Umbrich, J.: RDFS & OWL reasoning for linked data. *RDFS & OWL Reasoning for Linked Data* (2013), <http://ir.library.nuigalway.ie/xmlui/handle/10379/4385>
- [15] Hogan, A., Pan, J.Z., Polleres, A., Decker, S.: SAOR: template rule optimisations for distributed reasoning over 1 billion linked data triples. In: *The Semantic Web–ISWC 2010*, pp. 337–353. Springer (2010), http://link.springer.com/chapter/10.1007/978-3-642-17746-0_22
- [16] Hogan, A., Zimmermann, A., Umbrich, J., Polleres, A., Decker, S.: Scalable and distributed methods for entity matching, consolidation and disambiguation over linked data corpora. *Web Semantics: Science, Services and Agents on the World Wide Web* 10, 76–110 (2012), <http://www.sciencedirect.com/science/article/pii/S1570826811000813>
- [17] Horrocks, I., Patel-Schneider, P.: Reducing OWL entailment to description logic satisfiability. *Web Semantics: Science, Services and Agents on the World Wide Web* 1(4), 345–357 (2004), <http://www.sciencedirect.com/science/article/pii/S1570826804000095>
- [18] ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *Web Semantics: Science, Services and Agents on the World Wide Web* 3(2), 79–115 (2005), <http://www.sciencedirect.com/science/article/pii/S1570826805000144>
- [19] Imprialou, M., Stoilos, G., Grau, B.C.: Benchmarking Ontology-Based Query Rewriting Systems. In: *AAAI* (2012), <http://www.image.ntua.gr/papers/733.pdf>
- [20] Kikot, S., Kontchakov, R., Zakharyashev, M.: On (in) tractability of OBDA with OWL 2 QL. *CEUR Workshop Proceedings* (2011), <http://eprints.bbk.ac.uk/6635/>
- [21] Kiryakov, A., Ognyanov, D., Manov, D.: OWLIM—a pragmatic semantic repository for OWL. In: *Web Information Systems Engineering–WISE 2005 Workshops*. pp. 182–192. Springer (2005), http://link.springer.com/chapter/10.1007/11581116_19
- [22] Krötzsch, M.: OWL 2 Profiles: An introduction to lightweight ontology languages. In: *Reasoning Web - Semantic Technologies for Advanced Query Answering. Information Systems and Applications*, incl. Internet/Web, and HCI, vol. 7487, pp. 112–183. Springer-Verlag Berlin Heidelberg (2012), http://link.springer.com/chapter/10.1007/978-3-642-33158-9_4
- [23] Liu, C., Qi, G., Wang, H., Yu, Y.: Large scale fuzzy pd* reasoning using mapreduce. In: *The Semantic Web–ISWC 2011*, pp. 405–420. Springer (2011), http://link.springer.com/chapter/10.1007/978-3-642-25073-6_26
- [24] Lukasiewicz, T.: Expressive probabilistic description logics. *Artificial Intelligence* 172(6–7), 852–883 (Apr 2008), <http://www.sciencedirect.com/science/article/pii/S0004370207001877>

- [25] McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language Overview. W3c Recommendation (2004), <http://www.w3.org/TR/2004/REC-owl-features-20040210/>
- [26] Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language: Profiles (Second Edition). W3c Recommendation (2012), <http://www.w3.org/TR/2012/REC-owl2-profiles-20121211/>
- [27] Nute, D.: Defeasible logic. In: Web knowledge management and decision support, pp. 151–169. Springer (2003), http://link.springer.com/chapter/10.1007/3-540-36524-9_13
- [28] Pan, J.Z., Ren, Y., Jekjantuk, N., Garcia, J.: Reasoning the FMA Ontologies with TrOWL. In: ORE. pp. 107–113 (2013), http://www.cs.ox.ac.uk/isg/conferences/ORE2013/paper_18.pdf
- [29] Patel-Schneider, P.F., Hayes, P., Horrocks, I.: OWL Web Ontology Language Semantics and Abstract Syntax. W3c Recommendation (2004), <http://www.w3.org/TR/2004/REC-owl-semantics-20040210/>
- [30] Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artificial intelligence 48(1), 1–26 (1991), <http://www.sciencedirect.com/science/article/pii/000437029190078X>
- [31] Shadbolt, N., Hall, W., Berners-Lee, T.: The semantic web revisited. Intelligent Systems, IEEE 21(3), 96–101 (2006), http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1637364
- [32] Shi, H., Maly, K., Zeil, S.: A Scalable Backward Chaining-based Reasoner for a Semantic Web. International Journal On Advances in Intelligent Systems 7(1 and 2), 23–38 (2014), http://www.thinkmind.org/index.php?view=article&articleid=intsys_v7_n12_2014_3
- [33] Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide. W3c Recommendation (2004), <http://www.w3.org/TR/2004/REC-owl-guide-20040210/>
- [34] Stoilos, G., Stamou, G.B., Tzouvaras, V., Pan, J.Z., Horrocks, I.: Fuzzy OWL: Uncertainty and the semantic web. In: OWLED (2005), <http://www.image.ntua.gr/papers/398.pdf>
- [35] Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S.: Scalable nonmonotonic reasoning over RDF data using MapReduce. In: Proceedings of the Joint Workshop on Scalable and High-Performance Semantic Web Systems. pp. 75–90 (2012), <http://planet-data.eu/sites/default/files/publications/SSWS%2BHPCSW12.pdf>
- [36] Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S., McCluskey, T.L.: Large-scale parallel stratified defeasible reasoning (2012), <http://eprints.hud.ac.uk/15918>
- [37] Team, O.S.D.: OpenLink Virtuoso Universal Server: Documentation. Tech. rep. (2014), <http://docs.openlinksw.com/virtuoso/rdfsparqlrule.html>
- [38] Thomas, E., Pan, J.Z., Ren, Y.: TrOWL: Tractable OWL 2 reasoning infrastructure. In: The Semantic Web: Research and Applications, pp. 431–435. Springer (2010), http://link.springer.com/chapter/10.1007/978-3-642-13489-0_38

- [39] Urbani, J.: Scalable Distributed RDFS/OWL Reasoning using MapReduce. Master thesis, Amsterdam, Netherlands (2009)
- [40] Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F., Bal, H.: OWL reasoning with WebPIE: calculating the closure of 100 billion triples. In: *The Semantic Web: Research and Applications*, pp. 213–227. Springer (2010), http://link.springer.com/chapter/10.1007/978-3-642-13486-9_15
- [41] Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F., Bal, H.: WebPIE: A Web-scale parallel inference engine using MapReduce. *Web Semantics: Science, Services and Agents on the World Wide Web* 10, 59–75 (2012), <http://www.sciencedirect.com/science/article/pii/S1570826811000345>
- [42] Urbani, J., Kotoulas, S., Oren, E., Van Harmelen, F.: Scalable distributed reasoning using MapReduce. Springer (2009), http://link.springer.com/chapter/10.1007/978-3-642-04930-9_40
- [43] Urbani, J., Piro, R., van Harmelen, F., Bal, H.: Hybrid reasoning on OWL RL. *Semantic Web* 5(6), 423–447 (Jan 2014), <http://dx.doi.org/10.3233/SW-130120>
- [44] Urbani, J., Van Harmelen, F., Schlobach, S., Bal, H.: QueryPIE: Backward reasoning for OWL horst over very large knowledge bases. In: *The Semantic Web–ISWC 2011*, pp. 730–745. Springer (2011), http://link.springer.com/content/pdf/10.1007/978-3-642-25073-6_46.pdf
- [45] Xiao, G., Rezk, M., Rodríguez-Muro, M., Calvanese, D.: Rules and ontology based data access. In: *Web Reasoning and Rule Systems*, pp. 157–172. Springer (2014), http://link.springer.com/chapter/10.1007/978-3-319-11113-1_11
- [46] Zadeh, L.A.: Fuzzy sets. *Information and Control* 8(3), 338–353 (Jun 1965), <http://www.sciencedirect.com/science/article/pii/S001999586590241X>
- [47] Zou, Y., Finin, T., Chen, H.: F-owl: An inference engine for semantic web. In: *Formal Approaches to Agent-Based Systems*, pp. 238–248. Springer (2005), http://link.springer.com/chapter/10.1007/978-3-540-30960-4_16